Text Algorithms (6EAP)

Approximate Matching

Jaak Vilo

2018 fall

Exact vs approximate search

- In exact search we searched for a string or set of strings in a long text
- The we learned how to measure the similarity between sequences
- There are plenty of applications that require approximate search
- Approximate matching, i.e. find those regions in a long text that are similar to the query string
- E.g. to find substrings of S that have edit distacne < k to query string m.

Problem

- Given P and S – find all approximate occurrences of P in S

- **Reviews**
- P. A. Hall and G. R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381--402, 1980. ACM DL, PDF
- G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31--88, 2001. (Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999.) CiteSeer, ACM DL, PDF
- **Algorithms**
- S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM, 35(10):83--91, 1992. ACM DL PDF
- G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. Journal of the ACM, 46(3):395--415, 1999. CiteSeer, PDF
- A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. In Proc. 11th ACM-SIAM Symp. on Discrete Algorithms (SODA), pages 794--803, 2000. CiteSeer, ACM DL, PDF

- **Multiple approximate matching**
- R. Muth and U. Manber. Approximate multiple string search. In Proc. CPM'96, pages 75--86, 1996. CiteSeer, Postscript
- Kimmo Fredriksson - publications http://www.cs.uku.fi/~fredriks/publications.html
- **Applications**
- Udi Manber. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers and Security*, 15(2):171-- 176, 1996. CiteSeer, TR94-34, Postscript
- **Tools**
- Webglimpse - glimpse, agrep
  agrep for Win/DOS
  Original agrep
- **Links**
- Pattern Matching Pointers (Stefano Lonardi)
- Articles

Problem statement

- Let $S=s_1s_2...s_n \in \Sigma^*$ be a text and $P=p_1p_2...p_m$ the pattern. Let k be a pregiven constant.
- Main problems
- **k mismatches**
  - Find from S all substrings X, $|X|=|P|$, that differ from P at max k positsions (Hamming distance)
- **k differences**
  - Find from S all substrings X, where $D(X,P) \leq k$ (Edit distance)
- **best match**
  - Find from S such substrings X, that D(X,P) is minimal
- Distance D can be defined using one of the ways from previous chapters

Measure edit distance

	S=	A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C	A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
A	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
B	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
C	3	2	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
A	4	3	2	1	1	2	3	4	5	6	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
B	5	4	3	2	2	1	2	3	4	5	6	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C	6	5	4	3	2	2	1	2	3	4	5	6	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

Find approximate occurrences

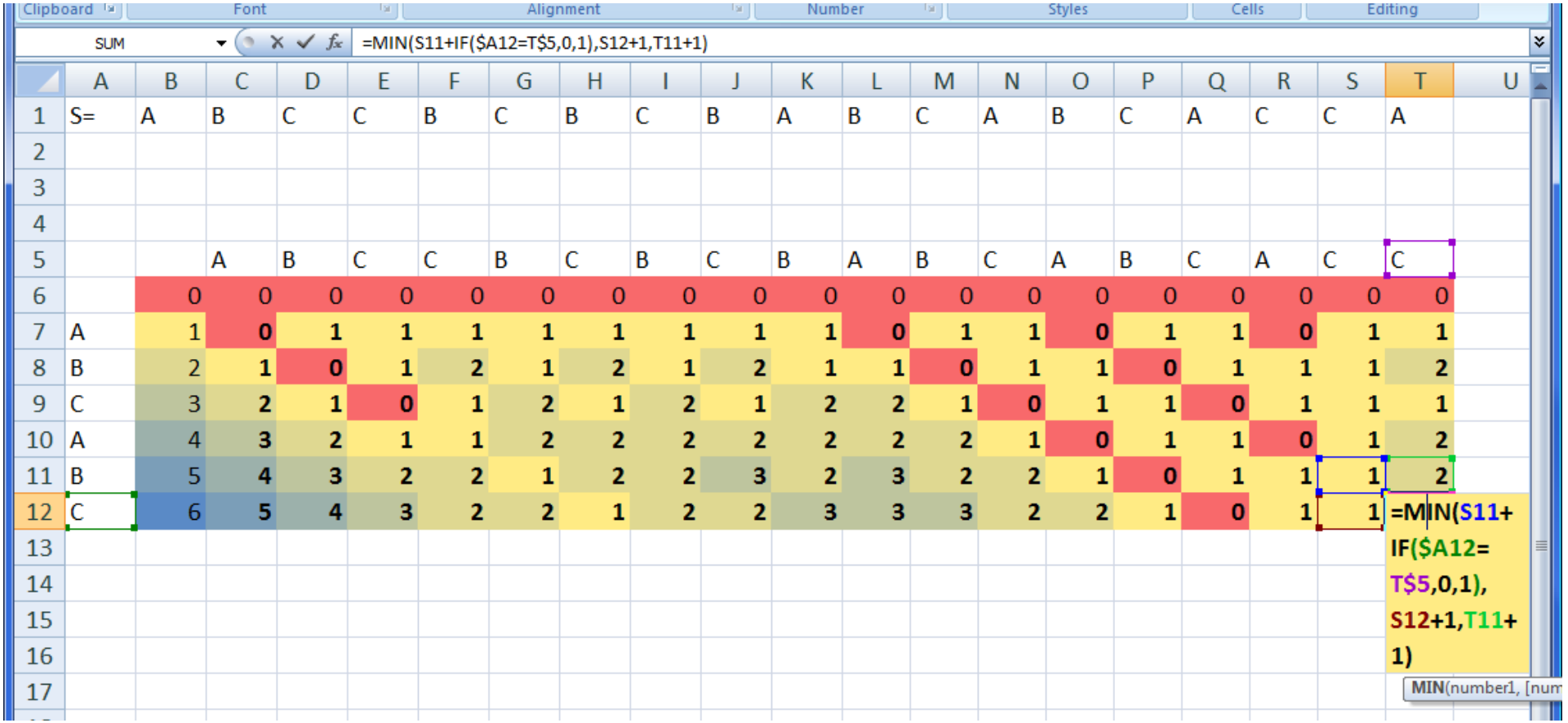
	S=	A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C	A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1	0	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1
B	2	1	0	1	2	1	2	1	2	1	1	0	1	1	0	1	1	1	2	1	0	1	2	1	2	1	2	1	1	0	1	1	0	1	1	1	2
C	3	2	1	0	1	2	1	2	1	2	2	1	0	1	1	0	1	1	1	2	1	0	1	2	1	2	1	2	2	1	0	1	1	0	1	1	1
A	4	3	2	1	1	2	2	2	2	2	2	2	1	0	1	1	0	1	2	1	2	1	1	2	2	2	2	2	2	2	1	0	1	1	0	1	2
B	5	4	3	2	2	1	2	2	3	2	3	2	2	1	0	1	1	1	2	2	1	2	2	1	2	2	3	2	3	2	2	1	0	1	1	1	2
C	6	5	4	3	2	2	1	2	2	3	3	3	2	2	1	0	1	1	1	2	2	1	2	2	1	2	2	3	3	3	2	2	1	0	1	1	1

Algorithm for approximate search, *k* edit operations

Input: P, S, k
Output: Approximate occurrences of P in S (with edit distance ≤ k)
for j=0 **to** m **do** $h_{j,0}$=j // Initialize first column
for i=1 **to** n **do**
    h0,i = 0
    for j=1 **to** m **do**
        $h_{j,i}$ = min($h_{i-1,j-1}$ + (if p_j==s_i then 0 else 1),
                $h_{i-1,j}$ + 1, $h_{i,j-1}$ + 1)
    if $h_{m,i}$ ≤ k Report match at i
    Trace back and report the minimizing path (from-to)

Example

	a	**b**	**r**	**a**	**c**	**a**	**d**	**a**	**b**	**r**	**a**
	0	0	0								
r	1	1	0								
a	2	1									
d	3	2									
a	4	3									



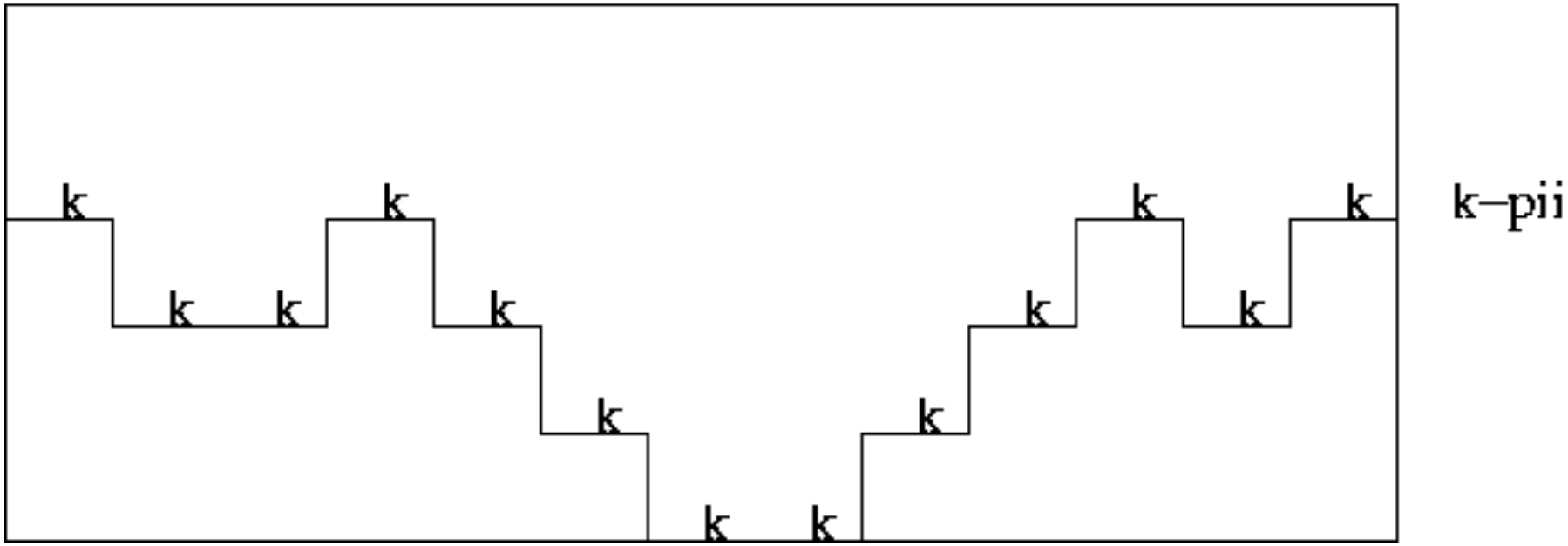
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	S=	A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C	A	
2																					
3																					
4																					
5			A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C	
6		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
7	A	1	0	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1	
8	B	2	1	0	1	2	1	2	1	2	1	1	0	1	1	0	1	1	1	2	
9	C	3	2	1	0	1	2	1	2	1	2	2	1	0	1	1	0	1	1	1	
10	A	4	3	2	1	1	2	2	2	2	2	2	2	1	0	1	1	0	1	2	
11	B	5	4	3	2	2	1	2	2	3	2	3	2	2	1	0	1	1	1	2	
12	C	6	5	4	3	2	2	1	2	2	3	3	3	2	2	1	0	1	1	=MIN(S11+IF($A12=T$5,0,1),S12+1,T11+1)	
13																					
14																					
15																					
16																					
17																					

- **Theorem** Lets assume that in the matrix h_{ij} the path that leads to the value h_{mj} in the last row starts from square h_{0r}. Then the edit distance $D(P, s_{r+1}s_{r+2}...s_j)=h_{mj}$, and h_{mj} is the minimal such distance for any substring starting before j'th position, $h_{mj}=\min\{ D(P,s_t s_{t+1}...s_j) \mid t \leq j \}$

- **Proof by induction**
- Every minimizing path starts from some value in the row 0
- Since it is possible to reach to the same result via multiple paths, then the approximate match is not always unique

- Time and space complexity O(mn)
- As n can be large, it is sufficient to keep the last m+k columns only, which can fully fit the full optimal path.
- Space complecity $O(m^2)$
- Or, one can keep just the single last column and in case of a match to recalculate the exact path.
- Space complecity O(m)
- If no need to find the pathm O(m)

- Diagonal lemma will hold
- If one nees to find only the regions with at most k edit operations, then one can restrict the depth of the calculations



- It suffices to compute until k-border
- Modified algorithm (home assignment) will work in average time O(kn)
- There are better methods which work in O(kn) atr the worst case.
- Landau & Vishkin (1988), Chang & Lampe (1991).

	S=	A	B	C	C	B	C	B	C	B	A	B	C	A	B	C	B	C	C	A	A	C	C	B	C	B	C	B	A	B	C	A	B	C	A	C	C
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	1	1	1	1	1	0	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	1	1	0	1	1	0	1	1
B	2	1	0	1	2	1	2	1	2	1	1	0	1	1	0	1	1	2	2	1	1	1	2	1	2	1	2	1	1	0	1	1	0	1	1	1	2
C	3	2	1	0	1	2	1	2	1	2	2	1	0	1	1	0	1	1	2	2	2	1	1	2	1	2	1	2	2	1	0	1	1	0	1	1	1
B	4	3	2	1	1	1	2	1	2	1	2	2	1	1	1	1	0	1	2	3	3	2	2	1	2	1	2	1	2	2	1	1	1	1	1	2	2
B	5	4	3	2	2	1	2	2	2	2	2	2	2	2	1	2	1	1	2	3	4	3	3	2	2	2	2	2	2	2	2	2	1	2	2	2	3
C	6	5	4	3	2	2	1	2	2	3	3	3	2	3	2	1	2	1	1	2	3	4	3	3	2	3	2	3	3	3	2	3	2	1	2	2	2
B	7	6	5	4	3	2	2	1	2	2	3	3	3	3	3	2	1	2	2	2	3	4	4	3	3	2	3	2	3	3	3	3	3	2	2	3	3
C	8	7	6	5	4	3	2	2	1	2	3	4	3	4	4	3	2	1	2	3	3	3	4	4	3	3	2	3	3	4	3	4	4	3	3	2	3
C	9	8	7	6	5	4	3	3	2	2	3	4	4	4	5	4	3	2	1	2	3	3	3	4	4	4	3	3	4	4	4	4	5	4	4	3	2
A	10	9	8	7	6	5	4	4	3	3	2	3	4	4	5	5	4	3	2	1	2	3	4	4	5	5	4	4	3	4	5	4	5	5	4	4	3
A	11	10	9	8	7	6	5	5	4	4	3	3	4	4	5	6	5	4	3	2	1	2	3	4	5	6	5	5	4	4	5	5	5	6	5	5	4
C	12	11	10	9	8	7	6	6	5	5	4	4	3	4	5	5	6	5	4	3	2	1	2	3	4	5	6	6	5	5	4	5	6	5	6	5	5
B	13	12	11	10	9	8	7	6	6	5	5	4	4	4	4	5	5	6	5	4	3	2	2	2	3	4	5	6	6	5	5	5	5	6	6	6	6

Improved average case

E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1-3):132-137, 1985.

```
1.  // Preprocessing
2.   for j=0..m do C[j]=j
3.   lact = k+1                              // last active row
4.  //  Searching
5.   for i=0..n
6.     pC=0;   nC=0            // previous and new column value
7.     for j=1 .. lact
8.       if S[i]==P[j] then nC=pC             //why?
9.       else
10.              if pC   < nC then nC = pC
11.              if C[j] < nC then nC = C[j]
12.              nC = nC+1
13.     pC = C[j]
14.     C[j] = nC
15.    while C[lact] > k do lact = lact -1
16.    if lact = m then report match at position i
17.    else lact = lact + 1
```

Ukkonen 1985; O(kn)


```
DP (p = p1p2...pm, T = t1t2...tn, k)
1.    Preprocessing
2.        For i ∈ 0...m Do Ci ← i
3.        lact ← k + 1 /* last active cell */
4.    Searching
5.        For pos ∈ 1...n Do
6.            pC ← 0, nC ← 0
7.            For i ∈ 1...lact Do
8.                If pi = tpos Then nC ← pC
9.                Else
10.                   If pC < nC Then nC ← pC
11.                   If Ci < nC Then nC ← Ci
12.                   nC ← nC + 1
13.               End of if
14.               pC ← Ci, Ci ← nC
15.           End of for
16.           While Clact > k Do lact ← lact − 1
17.           If lact = m Then report an occurrence at pos
18.           Else lact ← lact + 1
19.       End of for
```

Fig. 6.3. An $O(kn)$ expected time dynamic programming algorithm. Note that it works with just one column vector.

Four Russians technique

- This is a general technique that can be applied in different contexts
- It improves the speed of matrix multiplications
- Has been used for regular expression and approximate matching
- Let the column vector d*j = (d0j, ... , dmj) present the current state
- Lets preprocess the automaton from each state
- F(X,a)=Y, s.t. column vector X after reading character a becomes column vector Y.
- **Example:** Lets find **P=abc** approximate matches when there is at most 1 operation allowed.

Four Russians technique

- There are 13 different possibilities:

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**2**	**2**	**2**
0	**1**	**0**	**1**	**2**	**0**	**1**	**0**	**1**	**2**	**1**	**2**	**3**

- From each state compute possible next states for all characters a, b, c, and x (x not in P)
- The states with $d_{mj} \leq 1$ are final states.
- This can become too large to handle.
- Cut the regions into smaller pieces, use that to reduce the complexity.
- **Navarro and Raffinot** Flexible Pattern Matching in Strings. (Cambridge University Press, 2002). pp. 152 Fig 6.5.

Four-Russians version

Flexible Pattern Matching in Strings By Gonzalo Navarro, Mathieu Raffinot

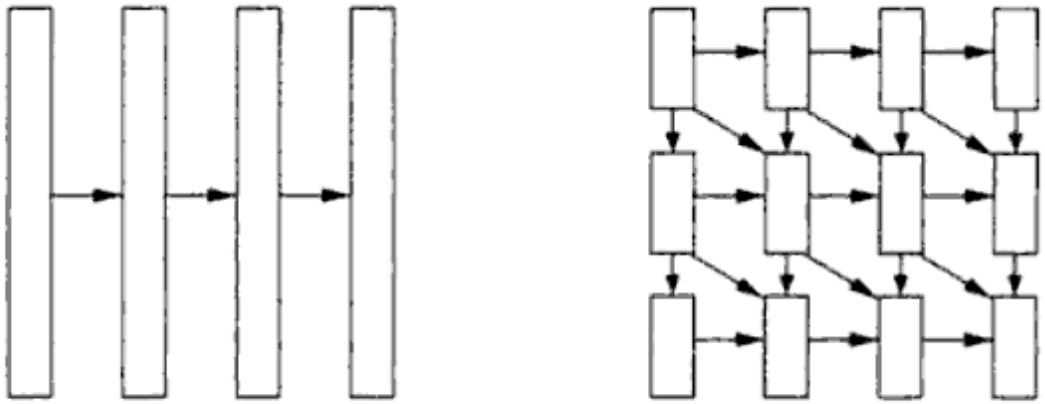


Fig. 6.5. On the left is the full DFA, where each column is a state. On the right is the Four-Russians version, where each region of a column is a state. The arrows show dependencies between consecutive regions.

12.7. The Four-Russians speedup

In this section we will discuss an approach that leads both to a theoretical and to a practical speedup of many dynamic programming algorithms. The idea, comes from a paper [28] by four authors, Arlazarov, Dinic, Kronrod, and Faradzev, concerning boolean matrix multiplication. The general idea taken from this paper has come to be known in the West as the Four-Russians technique, even though only one of the authors is Russian.[10] The applications in the string domain are quite different from matrix multiplication, but the general idea suggested in [28] applies. We illustrate the idea with the specific problem of computing (unweighted) *edit distance*. This application was first worked out by Masek and Paterson [313] and was further discussed by those authors in [312]; many additional applications of the Four-Russians idea have been developed since then (for example [340]).

NFA/DFA

- Create an automaton for matching a word approximately
- Allow 0,1,…n errors

cells. The cost of the diagonal arrows is 0 or 1, depending on whether the corresponding characters are equal (match) or different (substitution).

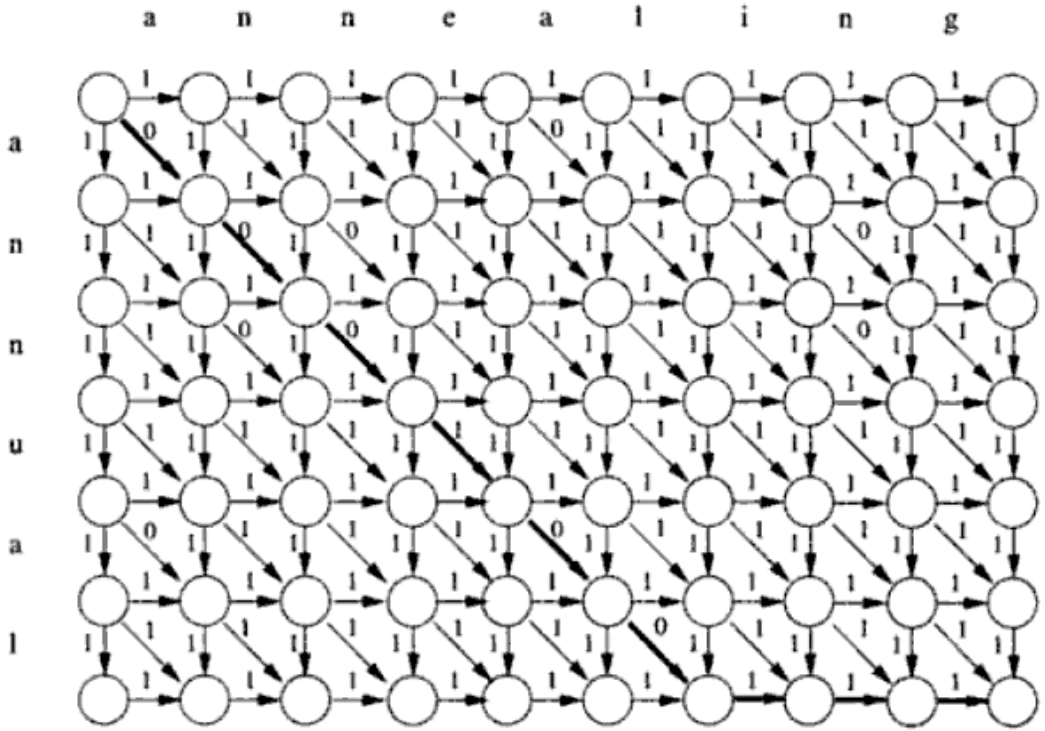


Fig. 6.18. Converting the edit distance problem into a shortest path problem. Bold arrows show the optimum path, of cost 4.

The edit distance problem can be converted into the problem of finding a shortest path from the upper left to the lower right node. If we are interested in approximate searching rather than in computing edit distance, then we assign zero cost to the horizontal arrows of the first row and consider minimum distances to every node of the last row.

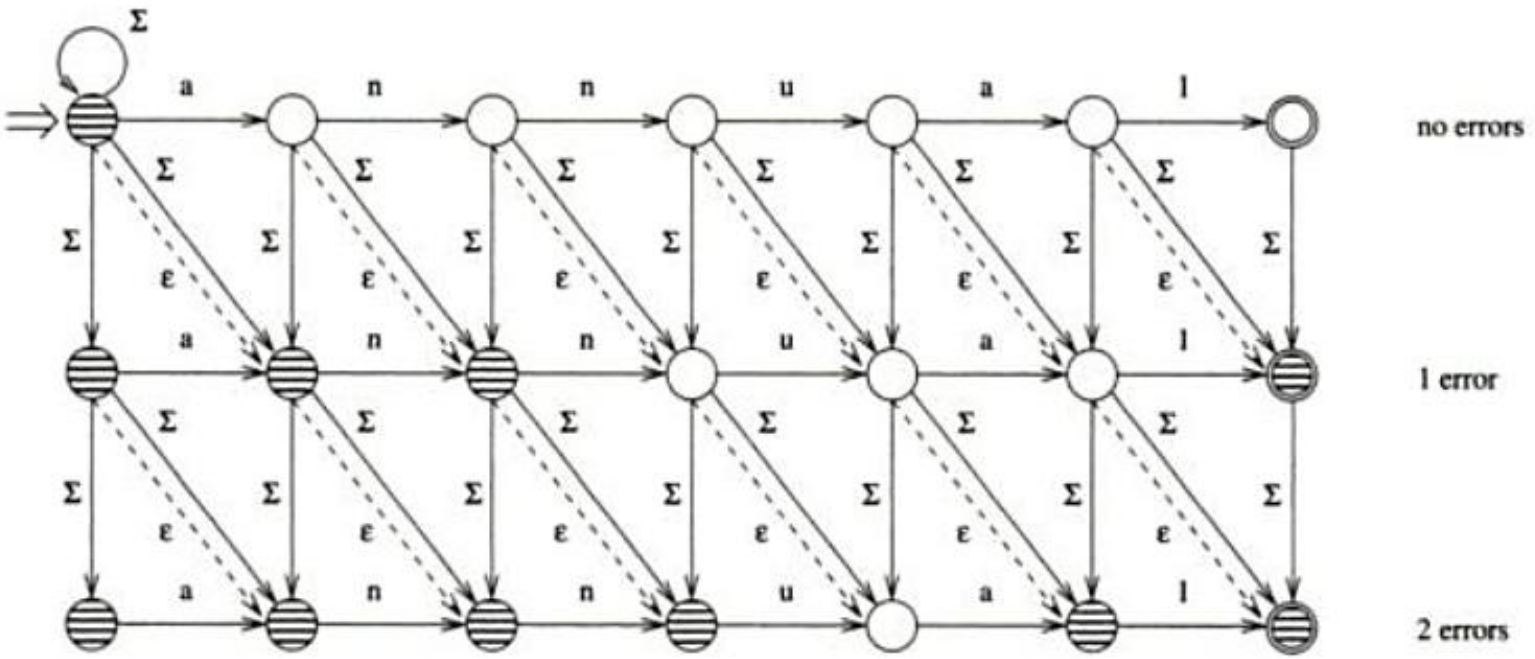


Fig. 6.4. An NFA for approximate string matching of the pattern `"annual"` with two errors. The shaded states are those active after reading the text `"anneal"`.

The original proposal of [Ukk85] was to make this automaton deterministic using the classical algorithm to convert an NFA into a DFA. This way, $O(n)$ worst-case search time is obtained, which is optimal. The main problem then becomes the construction and storage requirements of the DFA. An upper bound to the number of states of the DFA is $O(\min(3^m, m(2m|\Sigma|)^k))$ [Ukk85]. In practice, this automaton cannot be used for $m > 20$, and

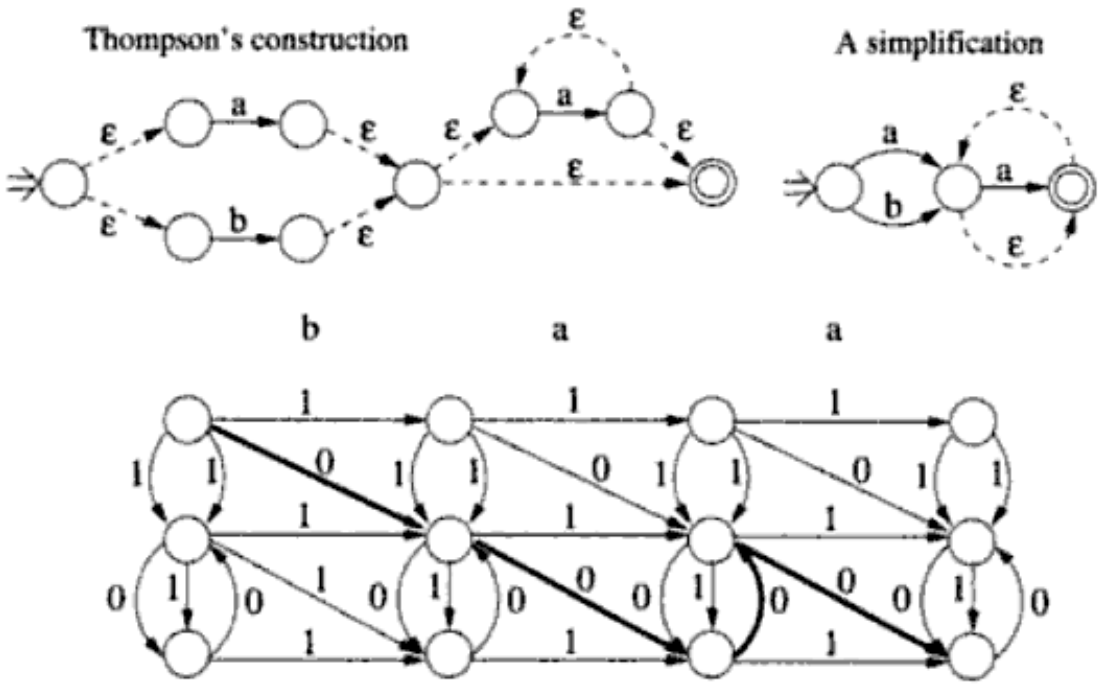


Fig. 6.19. The graph for the regular expression "(a|b)a*" on the text "baa". Bold arrows show an optimal path, of cost zero.

The idea of the shortest path can still be applied quite easily if the graph is acyclic, that is, if the regular expression does not contain the "*" or the "+" operator. On acyclic regular expressions we can find a topological order to evaluate the graph so as to find the shortest paths in overall time $O(mn)$. This requires Thompson's guarantee that there are $O(m)$ edges on

Regular expressions

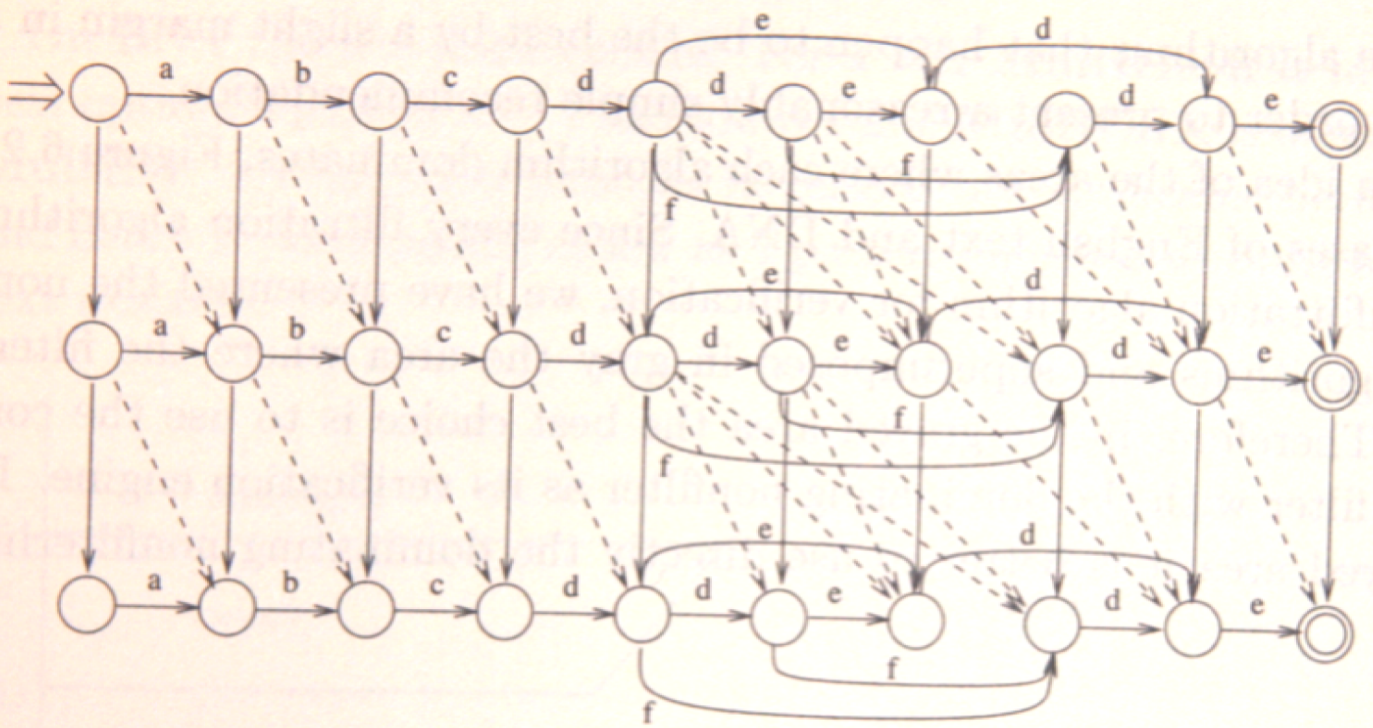


Fig. 6.20. Glushkov's NFAs for the regular expression "abcd(d|ε)(e|f)de" searched with two insertions, deletions, or substitutions. To simplify the figure, the dashed lines represent deletions and substitutions (i.e., they move by $\Sigma \cup \{\varepsilon\}$), while the vertical lines represent insertions (i.e., they move by Σ).

Filtering techniques

- q-gram (also k-mer, oligomer)
- (sub)string of length q
- Lets have a pattern P of length m
- Assume pattern P is rather long and k is small, find occurrences with at most k mismatches
- How long substrings of P must have an exact match?
- If mismatches are most evenly, then we get ~ m/k pieces

K mismatches

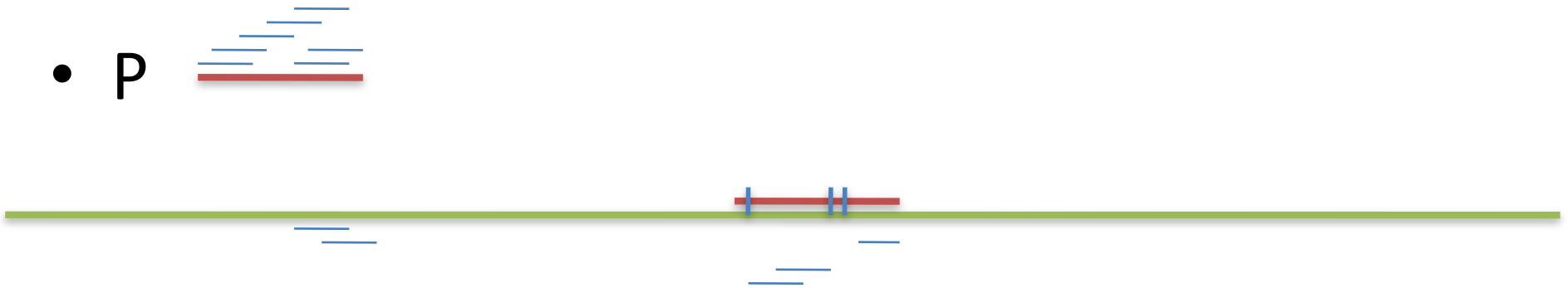
- K= 3
- P 
- For 3-mismatch match, at least one substring of length (m-3)/4 must occur exactly.

Filtering techniques with q-grams

- If P has k mismatches, then S must have at least one substring of P whose length is at least ⌈ (m-k)/k ⌉
- Filter for all possible q-mers where q is carefully selected.
  - Be careful with overlapping and non-overlapping q-grams.
  - If non-overlapping, then how long exact matches can we find?
- Use multiple exact matching O(n) (or sublinear) algorithms
- When an exact match of such substring is found, there is a possibility for an approximate overall match.
- Check for the actual match

Filter and verify!

- P



Filtering techniques cont.

- Lots of research on approximate matching using q-gram techniques
- Lots of times reinvented the wheel in different fields

Indexing using q-grams

- Filtering can also be used for indexing. E.g. index all q-grams and their matches in S.
- If one searches for P, first search for q-grams in index. If a sufficient nr of matches is found, then make the comparison to see if the match is real.
- Filtering should be efficient for cases where a high similarity match for a long pattern is looked for.
- This is like reverse index for texts:
- **word** doc_id:word_id doc_id:pos_id
- **word1** 1:5 7:9 167:987 ...
  word2 2:5 3:67 8:10 67:3 ...
  word3 3:5 5:67 7:10 16:3 ...
  ...
- Q: where do the word1 and word3 occur together?

Bit parallel search

- Can we use bit-parallelism for approximate search?

- T= lasteaed, P=aste

	l	a	s	t	e	a	e	d
	0	1	0	0	0	1		
	0	0	1	0	0	0		
	0	0	0	1	0	0		
	0	0	0	0	**1**	0		

Generalized patterns

- A generalized pattern $P=p_1p_2...p_m$ consists of generalized characters p_i such that each p_i represents a non-empty subset of alphabet Σ^*;
- $p_i = a$, $a \in \Sigma$
- p_i = # , "wildcard" (any nr any symbols)
- p_i = [group]; e.g.: [abc], [^abc], [a-h], ...
- $p_i = \neg C$; Characters from a set Σ-C.
- Example: [Tt][aeiou][kpt]#[^aeiou][mnr] matches Tekstialgoritm but not word tekstuur.
- Problem: Search for generalized patterns from text
- Compare to SHIFT-OR algorithm!

```
P= a[b-h]a¬a    // agrep  a[b-h]a[^a]
            paganamaa
a           110101
[b-h]       221011
a           332101
¬a          433210
```

zero at last row - exact match!

- What about mismatches?
- Mismatch if character does not belong to class defined by pattern. Unit cost 1.
- SHIFT-ADD - similar to SHIFT-OR, but instead of OR an ADD is used. (no insertions deletions on this example)

- (no insertions deletions on this example)

P= a[kpt]a¬a // agrep a[kpt]a[^a]

	p	a	g	a	n	a	m	a	a
	0	0	0	0	0	0	0	0	0
a	1	1	0	1	0	1			
[kpt]	2	2	1	**1**	2	1			
a	3	3	2	2	**1**	3			
¬a	4	3	3	2	2	**1**			

1 at last pos - match with 1 mismatch!

- Each value of matrix d_{ij} can be presented with b bits (4 bits allows values up to 16). Columns can be simple integers.

- Each value of matrix d_{ij} can be presented with b bits (4 bits allows values up to 16). Columns can be simple integers.
- $Bj = dmj\ 2^{b(m-1)} + dm\text{-}1,j\ 2^{b(m-1)} + \ldots\ d1j$. (d0j is always 0, can be omitted)
- When adding another integer, where 0 is on position i if the next char at j'th position belongs to a set represented by P_i and 1 otherwise.

- When adding another integer, where 0 is on position i if the next char at j'th position belongs to a set represented by P_i and 1 otherwise.

```
        010 001 000 001 011
+       001 001 000 000 001
        ---------------------------
=       011 010 000 001 100
```

- One needs to be very careful not to have overflow (111 + 001 = 1000).
- Shift by 3 positions == multiply by 8

010 001 000 001 011 * 8

= 001 000 001 011 000

Use multiple vectors, one for each k value

- One can also use several individual 1-bit vectors, each corresponds to different k
- Can be extended to mask out regions where mismatches are NOT allowed
- Can introduce wildcards of arbitrary length

Bit-parallelism

- Maintain a list of possible “states”
- Update lists using bit-level operations

Example

(note: least significant bit is left in this output)

Pattern = AC#T<GA>[TG]A length 7, # = .*

		7 6 5 4 3 2 1
CV[char]		
A 65	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	0 1 0 1 1 1 0
C 67	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 0 1
G 71	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 0 1 0 1 1 1
T 84	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 0 1 1 0 1 1
WILDCARD		
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 0 1
ENDMASK		
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 0 0 0 0 0
NO_ERROR		
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 1 1 0 0 0

0 – position is "active"

- **R[0] – vector for (so far) 0 mismatches**
- **R[1] – vector for (so far) 1 mismatch**
- **R[2] – vector for (so far) 2 mismatches**

- "**Minimum**" by bitwise AND
- If (even) one of the vectors has 0,
  then bitwise AND produces 0
  (which is smaller of 0 and 1, 1 and 0, 0 and 0)

- If both (or all) of the vectors have 1, then bitwise AND produces 1 (which is smaller of 1 and 1)

- How to get new values from old ones
- P[0] P[1] ... => R[0] R[1] ... R[0]
  - is min of three possibilities:

(P[i] shift 1) bitor CV[textchar]

// previously active, now match with character

(P[i] bitor WILDCARD)

// wildcard match – the same position remains active

(P[i-1] shift 1 bitor NO_ERROR)

// Previously 1 less errors (unless NO_ERROR allowed)

The algorithm

- R[i] in general is the minimum of 3 possibilities:

```
( P[i] shift 1 ) bitor CV[ textchar ] &        // match
( P[i] bitor WILDCARD ) &                      // wildcard
( P[i-1] shift 1 bitor NO_ERROR)               // mismatch

Last -- Add one mismatch unless errors not allowed
```

```
BPR (p = p1p2...pm, T = t1t2...tn, k)
1.    Preprocessing
2.        for c ∈ S Do B[c] <- 0m
3.        for j ∈ 1 ... m Do B[pj] <- B[pj] | 0m-j10j-1
4.    Searching
5.       for i ∈ 0 ... k Do Ri <- 0m-i1i
6.       for pos ∈ 1 ... n Do
7.           oldR <- R0
8.           newR <- ((oldR << 1) | 1) & B[tpos]
9.           R0 <- newR
10.          for i ∈ 1 ... k Do
11. newR <- ((Ri << 1) & B[tpos]) | oldR | ((oldR | newR) << 1)
12.              oldR <- Ri, Ri <- newR
13.          end of for
14.          If newR & 10m-1 <> 0 Then report an occurrence at pos
15.      End of for
```

```
public static void BPR(string pattern, string text, int errors)
        {
            int[] B = new int[ushort.MaxValue];
            for (int i = 0; i < ushort.MaxValue; i++) B[i] = 0;
            // Initialize all characters positions
            for (int i = 0; i < pattern.Length; i++)
            {
                B[(ushort)pattern[i]] |= 1 << i;
            }
            // Initialize NFA states
            int[] states = new int[errors+1];
            for(int i= 0; i <= errors; i++)
            {
                states[i] = (i == 0) ? 0 : (1 << (i - 1) | states[i-1]);
            }
            //
            int oldR, newR;
            int exitCriteria = 1 << pattern.Length -1;

            for (int i = 0; i < text.Length; i++)
            {
                oldR = states[0];
                newR = ((oldR << 1) | 1) & B[text[i]];
                states[0] = newR;

                for (int j = 1; j <= errors; j++)
                {
                    newR = ((states[j] << 1) & B[text[i]]) | oldR | ((oldR | newR) << 1);

                    oldR = states[j];
                    states[j] = newR;
                }

                if ((newR & exitCriteria) != 0)
                    Console.WriteLine("Occurrence at position {0}", i+1);

            }
      }
```

The Wu-Manber Algorithm of Text Searching Allowing Errors (1992)

$$\begin{cases} R_k^{[0]} & \leftarrow & 0^{m-k}1^k \\ R_0^{[j]} & \leftarrow & \left((R_0^{[j-1]} \ll 1) \mid 0^{m-1}1\right) \ \& \ B[t_j] \quad \text{(match)} \\ R_k^{[j]} & \leftarrow & \underbrace{\left((R_k^{[j-1]} \ll 1) \ \& \ B[t_j]\right)}_{\text{(match)}} \mid \underbrace{R_{k-1}^{[j-1]}}_{\text{(mismatch)}} \mid \underbrace{(R_{k-1}^{[j-1]} \ll 1)}_{\text{(insertion)}} \mid \underbrace{(R_{k-1}^{[j]} \ll 1)}_{\text{(deletion)}} \mid \underbrace{0^{m-k}1^k}_{\text{(init)}} \end{cases}$$

- Complexity: $O(ez)$
- Constraint: $m + e \leq w$
- Proposed by Wu and Manber, 1992 [Wu and Manber, 1992], implemented in the `agrep` software
- Allows 3 types of error: substitution, insertion, deletion
- Called Bit-Parallelism Row-wise (BPR), in [Navarro and Raffinot, 2002]

agrep

- S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM, 35(10):83--91, 1992. ACM DL PDF
- Insertions, deletions
- Wildcards
- Non-uniform costs for substitution, insertion, deletion
- Find best match
- Mask regions for no errors
- Record orientated, not line orientated

Agrep examples (from man agrep)

- **agrep -2 -c ABCDEFG foo**
  gives the number of lines in file foo that contain ABCDEFG within two errors.
- **agrep -1 -D2 -S2 'ABCD#YZ' foo**
  outputs the lines containing ABCD followed, within arbitrary distance, by YZ, with up to one additional insertion (-D2 and -S2 make deletions and substitutions too "expensive").
- **agrep -5 -p abcdefghij /usr/dict/words**
  outputs the list of all words containing at least 5 of the first 10 letters of the alphabet in order. (Try it: any list start- ing with academia and ending with sacrilegious must mean some- thing!)
- **agrep -1 'abc[0-9](de|fg)*[x-z]' foo**
  outputs the lines containing, within up to one error, the string that starts with abc followed by one digit, followed by zero or more repetitions of either de or fg, followed by either x, y, or z.
- **agrep -d '^From ' 'breakdown;internet' mbox**
  outputs all mail messages (the pattern '^From ' separates mail messages in a mail file) that contain keywords 'breakdown' and 'internet'.
- **agrep -d '$$' -1 ' ' foo**
  finds all paragraphs that contain word1 followed by word2 with one error in place of the blank. In particular, if word1 is the last word in a line and word2 is the first word in the next line, then the space will be substituted by a newline symbol and it will match. Thus, this is a way to overcome separation by a newline. Note that -d '$$' (or another delim which spans more than one line) is necessary, because otherwise agrep searches only one line at a time.
- **agrep '^agrep'**
  outputs all the examples of the use of agrep in this man pages.

- **Gene Myers: A fast bit-vector algorithm for approximate string matching based on dynamic programming** *Journal of the ACM (JACM),* Volume 46 , Issue 3 (May 1999). http://doi.acm.org/10.1145/316542.316550. PDF
- **Abstract**
- The approximate string matching problem is to find all locations at which a query of length *m* matches a substring of a text of length *n* with k-or-fewer differences.
- Simple and practical bit-vector algorithms have been designed for this problem, most notably the one used in agrep.
- These algorithms compute a bit representation of the current state-set of the k-difference automaton for the query, and asymptotically run in either O(nm/w) or O(nm log σ/w) time where w is the word size of the machine (e.g., 32 or 64 in practice), and σ is the size of the pattern alphabet.
- Here we present an algorithm of comparable simplicity that requires only O(nm/w) time by virtue of computing a bit representation of the relocatable dynamic programming matrix for the problem.
- Thus, the algorithm's performance is independent of k, and it is found to be more efficient than the previous results for many choices of k and small m.
- Moreover, because the algorithm is not dependent on k, it can be used to rapidly compute blocks of the dynamic programming matrix as in the 4-Russians algorithm of Wu et al.(1996).
- This gives rise to an O(kn/w) expected-time algorithm for the case where m may be arbitrarily large.
- In practice this new algorithm, that computes a region of the dynamic programming (d.p.) matrx w entries at a time using the basic algorithm as a subroutine is significantly faster than our previous 4-Russians algorithm, that computes the same region 4 or 5 entries at a time using table lookup.
- This performance improvement yields a code that is either superior or competitive with all existing algorithms except for some filtration algorithms that are superior when k/m is sufficiently small.
- Writing of an overview, implementing the algorithm and creating a useful tool could be a big topic for a BSc or MSc thesis.

Multiple approximate string matching

- How to find simultaneously the approximate matches for a set of words, e.g. a dictionary.
- Or a set of regular expressions, generalized patterns, etc.
- One can build automatons for sets of words, and then match the automatons approximately.
- Filtering approaches – if close enough, test
- Not many (good) methods have been proposed

- Overimpose NFA automata
- Filter on all (necessary) factors